High-Performance Computation of Radar Cross Sections: Parallelization of MATLAB Code[1]

D. R. Prabhu
The Ohio State University
PMB 193, 939-I Beards Hill Road
Aberdeen, MD 21001
E-mail: prabhu@arl.mil

T. Raju Damarla
Army Research Laboratory
2800 Powder Mill Road
Adelphi, MD 20783
E-mail: rdamarla@arl.mil

Abstract

A MATLAB code has been developed for computing scattered electromagnetic fields from, e.g., unexploded ordnance (UXOs), mines, and tanks. This code takes into account the lossy nature and frequency dependence of various soils based on a physical optics (PO) model. However, the simulation of scattered fields is computationally intensive: the number of computations is given by $N_f \times N_q \times N_f \times N_t \times N_p$, where N_f denotes the number of frequencies, N_q the number of elevation angles, N_f the number of azimuth angles, N_t the number of triangles in a patch model, and N_p the number of polarizations. Typically, in order to generate a synthetic aperture radar (SAR) image of a UXO at a given elevation angle, we need to compute the scattered fields at all frequencies for all azimuth angles ranging from $0°$ to $360°$, and the number of triangles in a patch model typically ranges from 3,000 to 10,000 for a UXO, depending on the fidelity of the model, which in turn depends on the highest frequency. Thus, for this example, the number of required computations is approximately 10^{10}. Furthermore, for large, complex objects such as tanks or trailers, the number of computations increases dramatically as N_t increases. Therefore, in order to generate the desired scattered fields in a reasonable time, it becomes highly desirable to parallelize the code.

The original serial code produced to solve this problem was written in MATLAB (m-file), and executed in approximately 3 hours on a SUN Ultra SPARC workstation. The MATLAB compiler was subsequently employed to convert the m-file to serial C code. The resulting serial C code was converted to parallel code by hand-insertion of calls to the Message Passing Interface (MPI) library. The computational workload in the loop that steps through the azimuth angles was distributed to multiple processors, resulting in coarse-grain parallelization. The parallel code produced identical outputs in only 9 minutes using eight processors of a SUN Enterprise 10000 machine at the U.S. Army

[1] This work was supported in part by the Programming Environment and Training (PET) program of the DoD High-Performance Computing Modernization Office.

Research Laboratory (ARL) Major Shared Resource Center (MSRC). The observed scalability of the code was close to linear.

Introduction

The Department of Defense (DoD) has a mission requirement to develop various techniques for the detection of unexploded ordnance (UXO) at various sites in the United States in order to identify UXO contamination and to facilitate remediation. Towards this end, the Army Research Laboratory (ARL) has been actively pursuing the use of an airborne ground–penetrating (GPEN) ultra–wideband (UWB) radar. ARL has built a UWB GPEN radar that is mounted on a telescopic boom that extends up to 50 m in order to simulate airborne radar. This UWB GPEN radar, which has a bandwidth of approximately 1.2 GHz (20-1200 MHz), is used to collect data from various test sites such as Yuma Proving Ground in Arizona and Eglin Air Force Base in Florida. In order to detect UXOs using physics-based phenomenology, a physical optics (PO) model [1–3] has been developed for computing scattered electromagnetic fields from various UXOs. This model takes into account the lossy nature and frequency dependence of various soils, which should allow for more accurate modeling of UXOs in a variety of realistic field conditions. The computation of the scattered fields requires the development of high–fidelity models (triangular patch models) of UXOs, and the simulation of the scattered fields using a PO simulator for various azimuth angles, elevation angles, and different frequencies (20-2000 MHz). Figures 1a and 1b show the triangular patch model of a 155–mm artillery shell (UXO155) and the resulting scattered fields. Time–domain pulses reflected from a UXO are obtained by computing the inverse Fourier transform of the product of the simulated fields and the spectrum of the transmitted radar pulse. These time–domain pulses are subsequently used to generate SAR images.

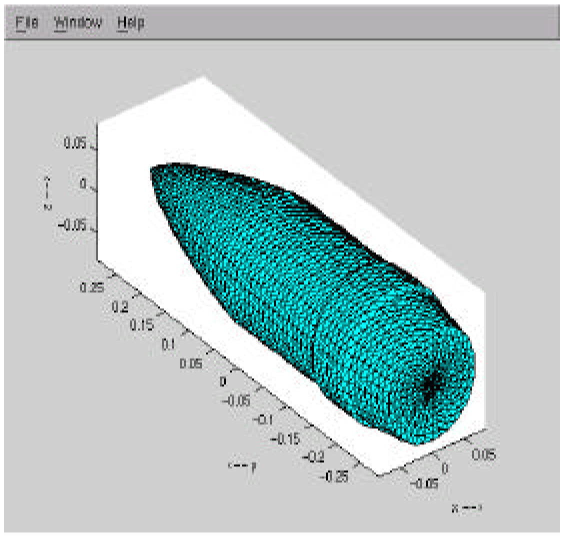

Figure 1a: High–fidelity patch model of UXO155

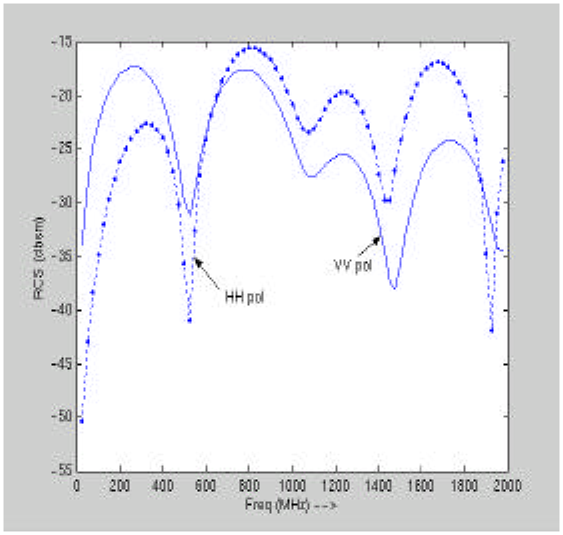

Figure 1b: RCS characteristics of UXO155

For each elevation and azimuth angle, the PO simulator computes currents induced by an incident radar-transmitted plane wave on each triangular patch of the UXO. These currents are computed for all frequencies in the radar bandwidth, and for all four polarizations of the incident and scattered waves, namely, VV, VH, HV, and HH, where V denotes vertical polarization and H denotes horizontal polarization. As a result, the simulation of the composite scattered fields is computationally expensive: the number of computations is given by $N_f \times N_q \times N_f \times N_t \times N_p$, where N_f denotes the number of frequencies, N_q the number of elevation angles, N_f the number of azimuth angles, N_t the number of triangles in a patch model, and N_p the number of polarizations. For the generation of SAR images of a UXO, one needs to compute electromagnetic scattering fields at azimuth angles ranging from $0°$ to $360°$ for each elevation angle. The computation of N_t, the number of patches in a high–fidelity model, depends on the size of the UXO, and the highest frequency at which the scattered fields are computed. Typically, the side of a triangle in a patch model is roughly $0.1\, l$, where l denotes the wavelength. Consequently, N_t typically ranges from 3000 to 10000 for a UXO, and N_t is an order or two larger for tanks, trailers, etc. As can be seen from this straightforward analysis, in order to generate the electromagnetic scattering fields in a reasonable time, it becomes highly desirable to parallelize the code.

The PO simulator code was developed in MATLAB [4], which has been identified as one of the key higher level programming languages for algorithm development in signal/image processing. Core MATLAB, along with numerous toolboxes for specialized applications, provides a rich collection of functions and visualization tools for rapid code development and prototyping. Unfortunately, MATLAB is inherently serial, and while parts of MATLAB that are linked to vendor provided libraries may execute in multi-threaded mode, overall, the result is only a marginal parallelization of user–developed code.

This paper presents details of parallelization of the PO simulator code. The MATLAB compiler was employed to convert the MATLAB code to serial C code. The resulting serial code was subsequently parallelized through the hand-insertion of calls to the MPI library. Studies performed on the resulting parallel code showed close to linear scalability.

Serial MATLAB Code

The flowchart in Figure 2 shows the structure of the MATLAB code for the PO simulator. Calculating the electromagnetic fields consists of computing the following field equation [1,3] for vertical polarization:

$$E_q = -\frac{jwme^{-jbr}}{4pr} \iint_s \left(J_x \cos q \cos f + J_y \cos q \sin f - J_z \sin q \right) e^{-jb\left(x'\sin q \cos f + y'\sin q \sin f + z'\cos q\right)} ds ,$$

where r denotes the radial distance from the radar to the target, w denotes the radial frequency, m denotes the permeability, $b = w\sqrt{me}$, e denotes the permittivity, vector $J = \left(J_x, J_y, J_z\right)$ denotes the current in a patch, $r' = (x', y', z')$ denotes the vector from the origin to the centroid of a triangle patch, q denotes the elevation angle, and f denotes the azimuth angle. A similar equation is used to compute the horizontal polarization. The code, which is well written in MATLAB with vectorization to rapidly compute the currents on all triangular patches, executes in about 3 hours on a SUN UltraSPARC workstation equipped with a 300–MHz CPU and 256 MB of main memory. The execution time was reduced to 2 hours when the code was ported to a SUN E10K at the ARL MSRC and executed under MATLAB.

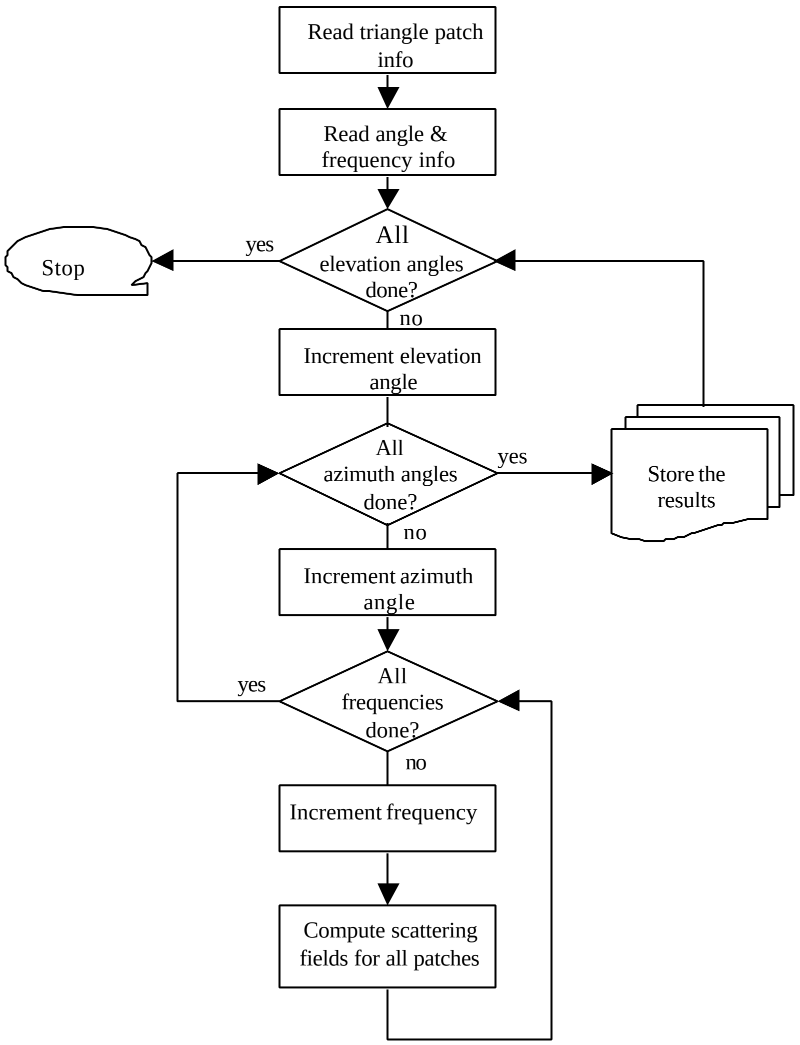

Figure 2: Structure of MATLAB serial code

Parallelization of MATLAB Code

The serial MATLAB code in m-file form was converted to a parallel, standalone MPI-based executable. A schematic of the methodology employed is presented in Figure 3.

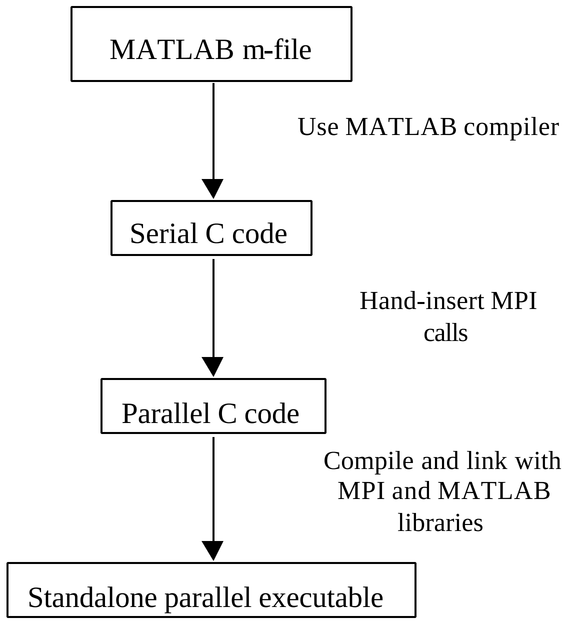

Figure 3: Conversion of serial MATLAB code to a standalone executable

The conversion involved the following steps:

Conversion of MATLAB code (m-file) to serial C code: The MATLAB compiler (version 2) was employed to convert the m-file into serial C code. This conversion was achieved in a straightforward manner by invoking the MATLAB compiler (*mcc*) inside MATLAB. The MATLAB compiler converts each variable in MATLAB to an *mxArray* structure in C, which contains the following information: pointers to the values of the real and imaginary parts, array size of the variable (all variables in MATLAB are arrays), the name of the variable in the MATLAB workspace, etc. Mathematical operations and assignments in MATLAB are converted to procedure calls in C. For example, the statement "a = b+c" in MATLAB is transformed to "mlfAssign (&a, mlfPlus(b, c))" in the resulting C code, where a, b, and c are *mxArray* structures.

At this point, we note that it is critical to obtain good serial performance before any parallelization work is undertaken. For example, the highest level of optimization should be used that results in the best performance and yields correct results. In addition, compiler flags that fully exploit the processor architecture and available caches on the compute platform should be used. Additionally, linking to a fast math library can often yield a significant speed-up, but the precision of the results could be affected.

Accordingly, the resulting C code was optimized for serial performance. The accuracy and the precision of the results of the standalone executable were verified by comparison to the original MATLAB code results. Identical outputs were obtained with an execution time of about 1 hour, yielding a speed-up factor of 2 without any parallelization.

Parallelization of serial C code: The original code requires the following input information: a) angles and their ranges for a specific simulation run, b) material properties, c) coordinates of patches defining the test object, and d) connectivity information to define the test object geometry. The required information is organized in three input files. The MATLAB-generated C code uses special functions to read input values and to assign them to *mxArray* type variables.

All computing processes need to be able to access all input data during execution. The root process typically reads data files and broadcasts needed information to other processes. However, this approach has disadvantages for MATLAB−generated C code. MPI-based sends and receives of *mxArray* structures involve the extensive use of user-defined data types, which requires writing a significant amount of additional code. It is also inefficient to extract individual object values from the *mxArray* structures for broadcast to all processes during parallel execution. Moreover, each process in turn needs to receive individual values and reencapsulate them into *mxArray* structures. Consequently, we decided not to use MPI-based communication for broadcasting the input data. Instead, the three input data files were replicated to provide each process with its own set of data files, resulting in simpler code that was easier to debug and maintain. This technique works for typical input files, which are about 100 kilobytes in size. This approach would need to be reevaluated if disk space is constrained and/or if test objects have complex geometries, resulting in huge data files.

The loop that steps through the azimuth angle, which is shown in Figure 2, was parallelized using MPI. Calls to MPI functions were hand-inserted into the serial C code, resulting in coarse-grain parallelization. Since each iteration of the azimuth angle loop was independent, the computational load of the loop was distributed across multiple processes in a straightforward manner. This process can be described by the following series of steps.

- N parallel processes are created on N processors
- The root process (process with rank 0) creates N replicas of each input file, and assigns a unique name to each copy; numbers 0 through $N-1$ are appended to the original name of each input file.
- Each process reads its corresponding set of input files
- The root process deletes all replicas of input files
- The root process computes the work load distribution, and assigns each process a share of work. For this application, each process is assigned start and end values for the azimuth angle loop variable.
- Each process computes its assigned part of the loop and writes output files to disk.

Compilation and linking of parallel C code: The modified C code with MPI calls was compiled and linked with the MPI library (libmpi), and executed on multiple processors. The following MATLAB-related libraries were also linked in for standalone execution: libmmfile, libmcc, libmatlb, libmat, and libmx.

Results

The execution time of the original code was approximately 3 hours on a SUN Ultra SPARC workstation. As previously mentioned, the execution time reduced to 2 hours when the code was ported to a SUN E10K at the ARL MSRC and executed under MATLAB. In standalone mode, the serial optimized code produced identical outputs in about 1 hour. In contrast, the standalone parallel code produced identical outputs in 9 minutes using 8 processors. A plot of the execution time of the parallel code vs. number

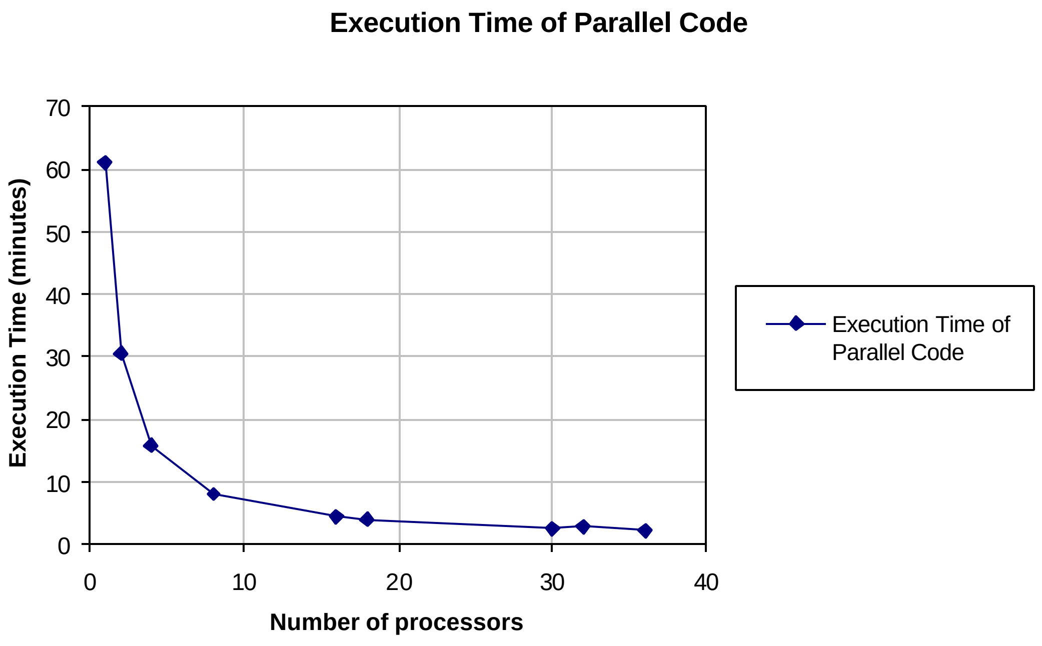

Figure 4: Execution time of parallel radar cross-section computation code. Plot shows execution time (wall clock time, minutes) vs. number of processors

of processors is presented in Figure 4. A plot of the resulting speed-up vs. number of processors is shown in Figure 5. Both plots are based on results obtained on a nondedicated 64–processor SUN E10K HPC with UltraSPARC II 400–MHz CPUs and 64 GB of main memory. A linear reference line and the ideal speed-up curve are superimposed on the speed-up plot in Figure 5.

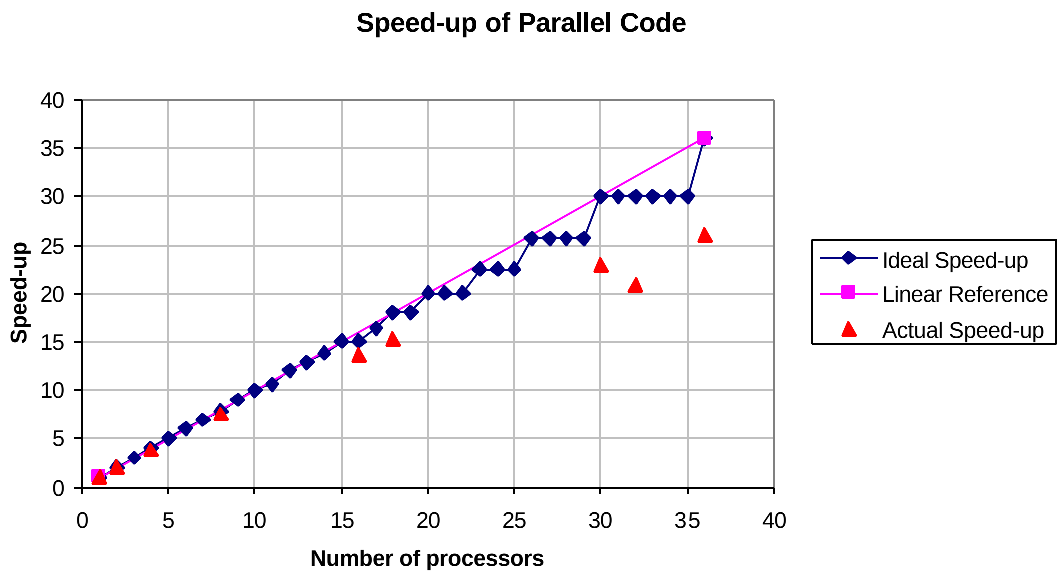

Figure 5: Speed-up of parallel radar cross-section computation code. Plot shows speed-up vs. number of processors

The ideal speed-up curve is computed with the following “ideal” assumptions: a) zero process startup times, b) infinite communication and file I/O bandwidths, and c) equal computation times for each iteration of the azimuth angle loop. Note that the staircase appearance of the ideal curve is due to the coarse-grain parallelization of a finite number of loop iterations. When the total number of iterations (180 for this problem) is a multiple of the number of processors, each processor is assigned an equal number of iterations in the workload distribution. The resulting speed-up, under assumed ideal conditions, is linear. However, when the total number of iterations is not a multiple, the distribution of workload is unequal. Processors that have larger computational loads determine the time to completion, resulting in sublinear ideal speed-up.

In reality, process startup times, limited communication bandwidth, and file I/O overhead can have significant effects on execution times, depending on the architecture. Figure 5 clearly shows such effects, when the number of processors is increased from 30 to 32. As seen from the ideal speed-up curve, there is no computational benefit associated with the increase. The observed speed-up drops, instead of remaining constant.

The resulting speed-up is slightly below the ideal speed-up curve for the given problem size, using up to 36 processors. Increasing the number of processors beyond 36 is not recommended for this problem size.

Summary and Conclusions

The improved, parallelized code has had a significant impact on ground–penetrating ultra–wideband radar system simulations at the Army Research Laboratory. The parallel code enables rapid computation of a) radar cross-sections at higher angular resolutions, b) radar cross-sections of objects with complex geometries, and c) more intensive additional computations within the loop.

There are other ways of parallelizing serial MATLAB code. Integrated Sensors, Inc. (ISI), has developed a software package called RTExpress [5] that converts code written in MATLAB to C code, compiles with MPI libraries, and supports execution on multiple processors with minimal input from the user. Parallelization is achieved by replacing MATLAB functions with MPI-based parallel versions. Over 90% of core MATLAB functions are supported. RTExpress has been installed on the SUN E10Ks at the ARL MSRC, and will be used to parallelize the radar cross-section code. Another alternative is to use NetSolve [6], developed at the University of Tennessee, which enables users to access network distributed computational resources. Using the MATLAB interface to NetSolve, application programs are executed on a UNIX workstation or a PC, but computationally intensive parts of the code are executed (in serial or parallel) on remote HPC NetSolve servers. The use of OpenMP directives is also slated for investigation on the SUN E10K and other shared memory machines such as SGI O2Ks.

Acknowledgments

This work was supported in part by the Programming Environment and Training (PET) program of the DoD High Performance Computing Modernization Office. The authors wish to thank Prof. Stan Ahalt for insightful discussions.

References

1. T. Damarla, "A Fast Algorithm for Computing Scattered Fields using Physical Optics Equivalent Approximation in Half Space," Technical Report, Army Research Laboratory, Adelphi, MD.
2. Anders Sullivan, Raju Damarla, Norbert Geng, Yanting Dong, and Lawrence Carin, "Electromagnetic Modeling of Surface and Buried Unexploded Ordnance (UXO)," in the Proc. of ARL-FEDLAB Symposium, 21-23 March 2000, College Park, MD.
3. C. A. Balanis, "Advanced Engineering Electromagnetics," published by John Wiley & Sons, 1989.
4. MATLAB, http://www.mathworks.com
5. RTExpress, http://www.rtexpress.com
6. Henri Casanova and Jack Dongarra, "NetSolve: A Network Server for Solving Computational Science Problems," Technical Report No. CS-95-313, University of Tennessee, November 1995.